10 Million Smart Meter Data with Apache HBase

5/31/2017
OSS Solution Center
Hitachi, Ltd.
Masahiro Ito

Open Source Summit Japan 2017

Who am I?

HITACHI
Inspire the Next

- Masahiro Ito
  - ➢ Software Engineer at Hitachi, Ltd.
  - ➢ Focus on development of Big Data Solution with Apache Hadoop and its related OSS.
  - ➢ Mail: masahiro.ito.ph@hitachi.com
  - ➢ Book and Web-articles (in Japanese)
    - Apache Spark ビッグデータ性能検証
（Think IT Books）
    - **ユースケースで徹底検証！**
HBaseでIoT時代のビッグデータ管理機能を試す
      - – https://thinkit.co.jp/series/6465

Agenda

1. Motivation
2. What is NoSQL?
3. Overview of HBase architecture
4. Performance evaluation with 10 million smart meter data
5. Summary

1. Motivation

Motivation

- The internet of things (IoT) and NoSQL
  - ➢ Various sensor devices generate large amounts of data.
  - ➢ NoSQL has higher performance and scalability than RDB.
  - ➢ HBase is one of NoSQL.

- Is HBase suitable for sensor data management?
  - ➢ HBase seems to be suitable for managing time series data such as sensor data.
  - ➢ I will introduce the result of performance evaluation of HBase with 10 million smart meter data.

2. What is NoSQL?

NoSQL (Not only SQL)

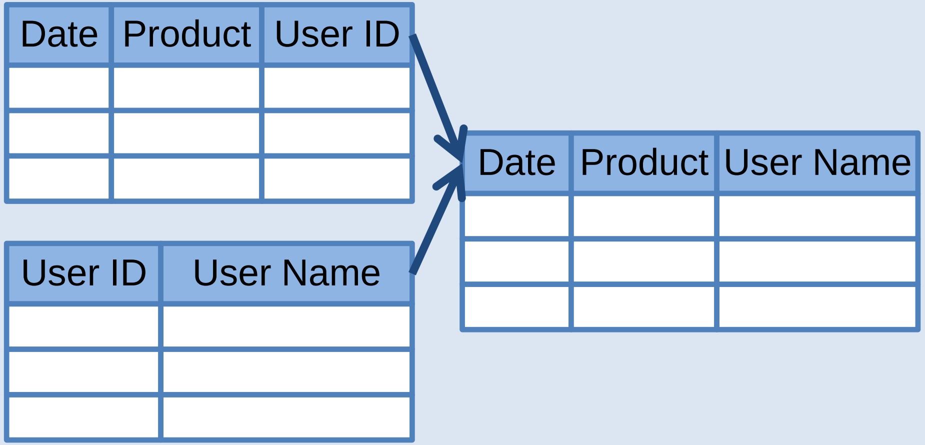
- NoSQL refers to databases other than RDB (Relational DataBase).

- Motivations of NoSQL include:
  - ➢ More flexible data model (not tabular relations).
  - ➢ High performance and large disk capacity.
    - With simpler "horizontal" scaling to clusters of machines.
  - ➢ etc.

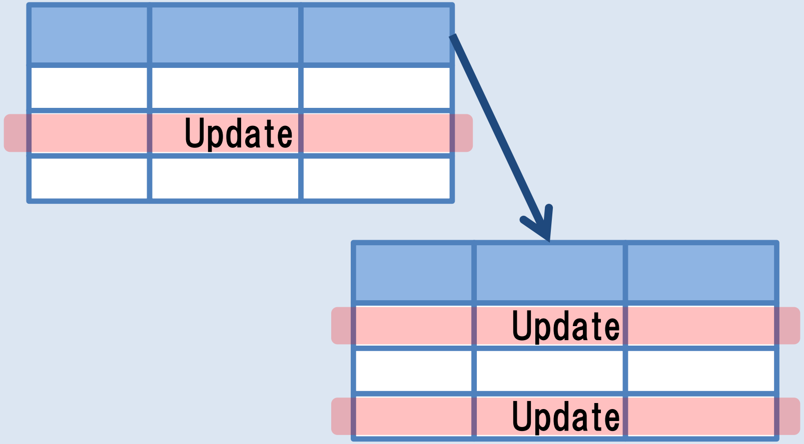
- NoSQL databases are increasingly used in big data and real-time web applications.

Features of RDB

Relational model



ACID Transaction



- Table format (tabular relations)
- SQL interface
  - Supports complex queries

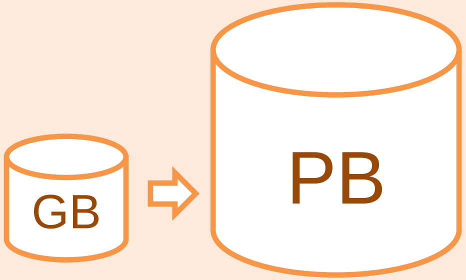
- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

3 Vs of Big Data: Challenges of RDB for big data

HITACHI
Inspire the Next

Volume

Need to manage large amount of distributed data.

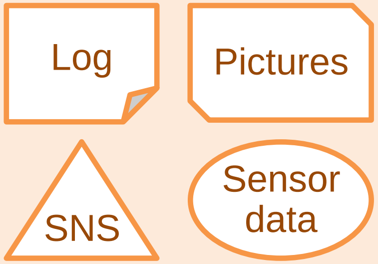


Velocity

Need to process large number of requests in real time.

Variety

Need to manage data of various structures.



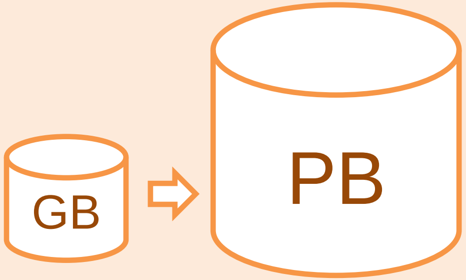
RDB

Volume	Velocity	Variety
Transaction control over distributed data is difficult.	Exclusive control of transaction is overhead.	It is incompatible with the predefined table.

3 Vs of Big Data: Challenges of RDB for big data

Volume

Need to manage large amount of distributed data.

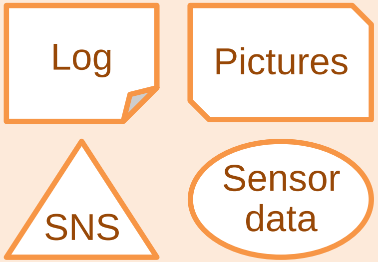


Velocity

Need to process large number of requests in real time.

Variety

Need to manage data of various structures.



RDB

Volume	Velocity	Variety
Transaction control over distributed data is difficult.	Exclusive control of transaction is overhead.	It is incompatible with the predefined table.

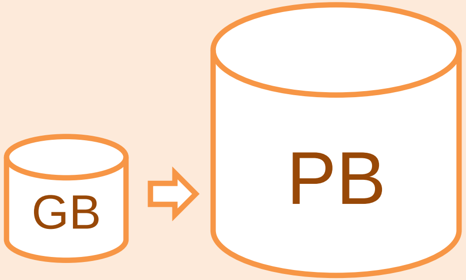
NoSQL

Limiting the scope of transaction control makes it possible to improve performance and disk capacity with scale out.

3 Vs of Big Data: Challenges of RDB for big data

Volume

Need to manage large amount of distributed data.

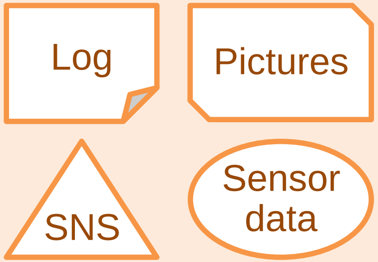


Velocity

Need to process large number of requests in real time.

Variety

Need to manage data of various structures.



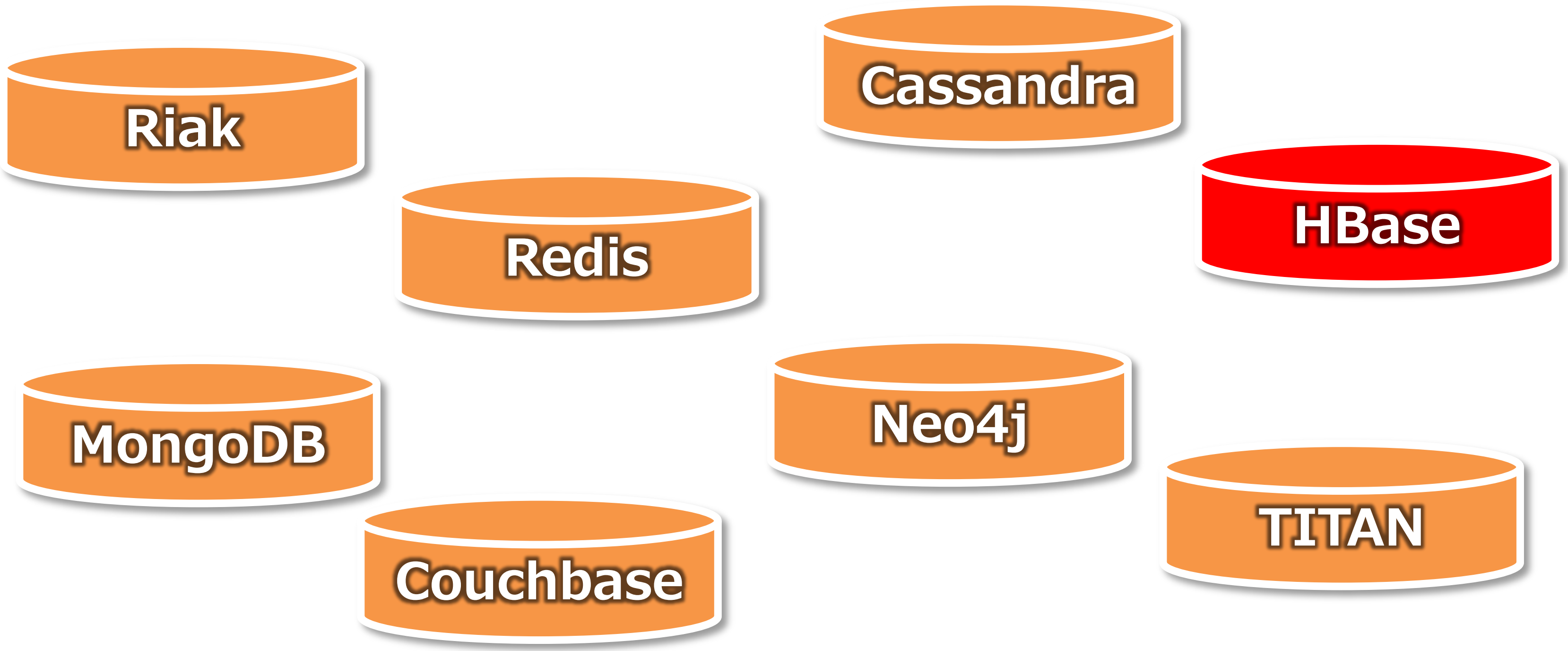
RDB

- Volume: Transaction control over distributed data is difficult.
- Velocity: Exclusive control of transaction is overhead.
- Variety: It is incompatible with the predefined table.

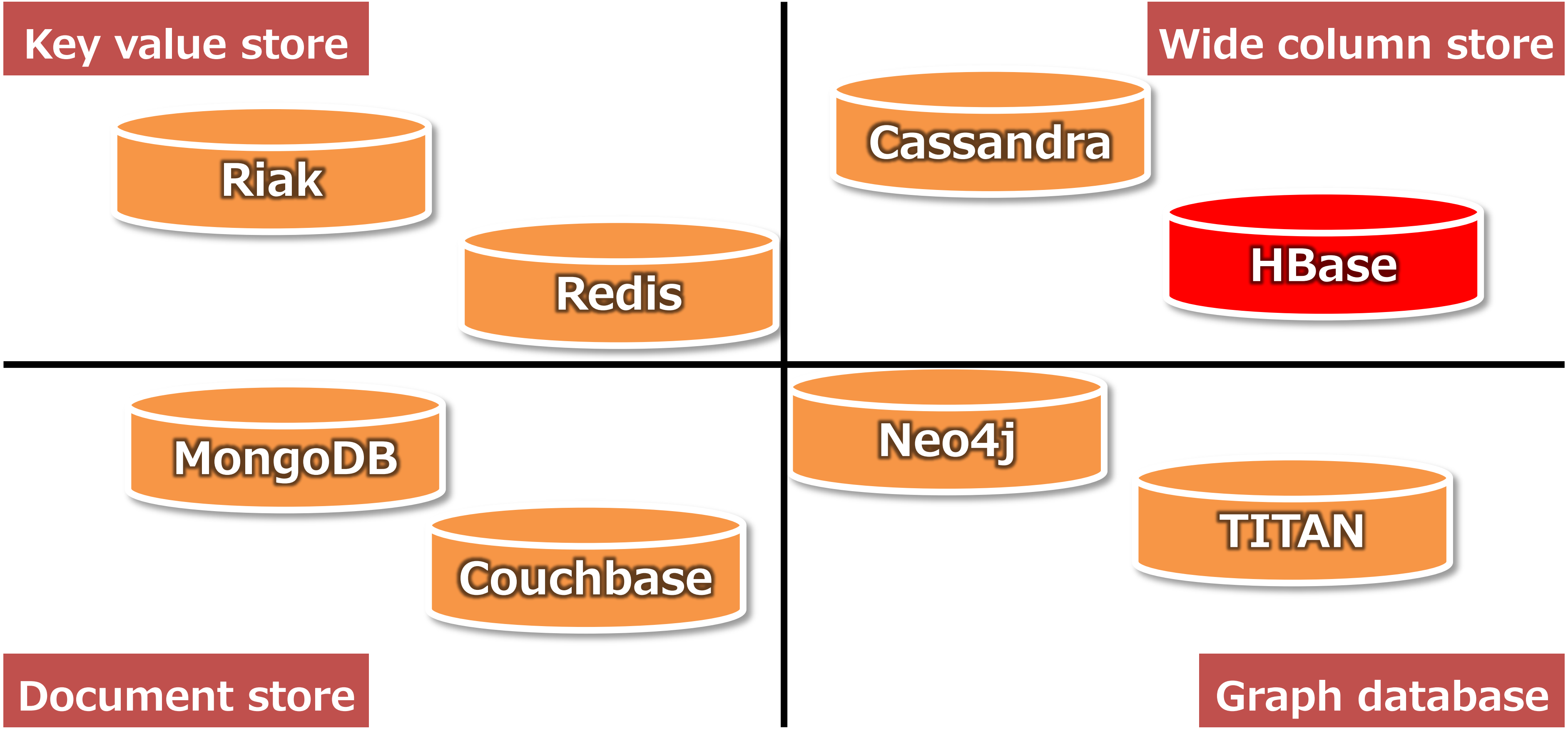
NoSQL

- Volume / Velocity: Limiting the scope of transaction control makes it possible to improve performance and disk capacity with scale out.
- Variety: Adopted flexible data structure other than table.

There are lots of NoSQL in the world（many others）



NoSQL is generally classified by data model



NoSQL is generally classified by data model

Key value store

Low latency access with simple data structure.

Key	Value

Wide column store

Each row has different number of columns.

Key	Value	Value	Value

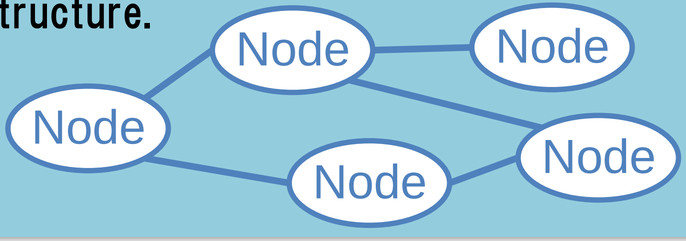
Document store

Store structure data such as JSON.

Key	Document
001	{ ID: 001 User: { Name: “Engineer” } }

Graph database

Represent relationship between data as graph structure.



3. Overview of HBase architecture

HBase overview

Hitachi Inspire the Next

- HBase is distributed, scalable, versioned, and non-relational (wide column type) big data store.

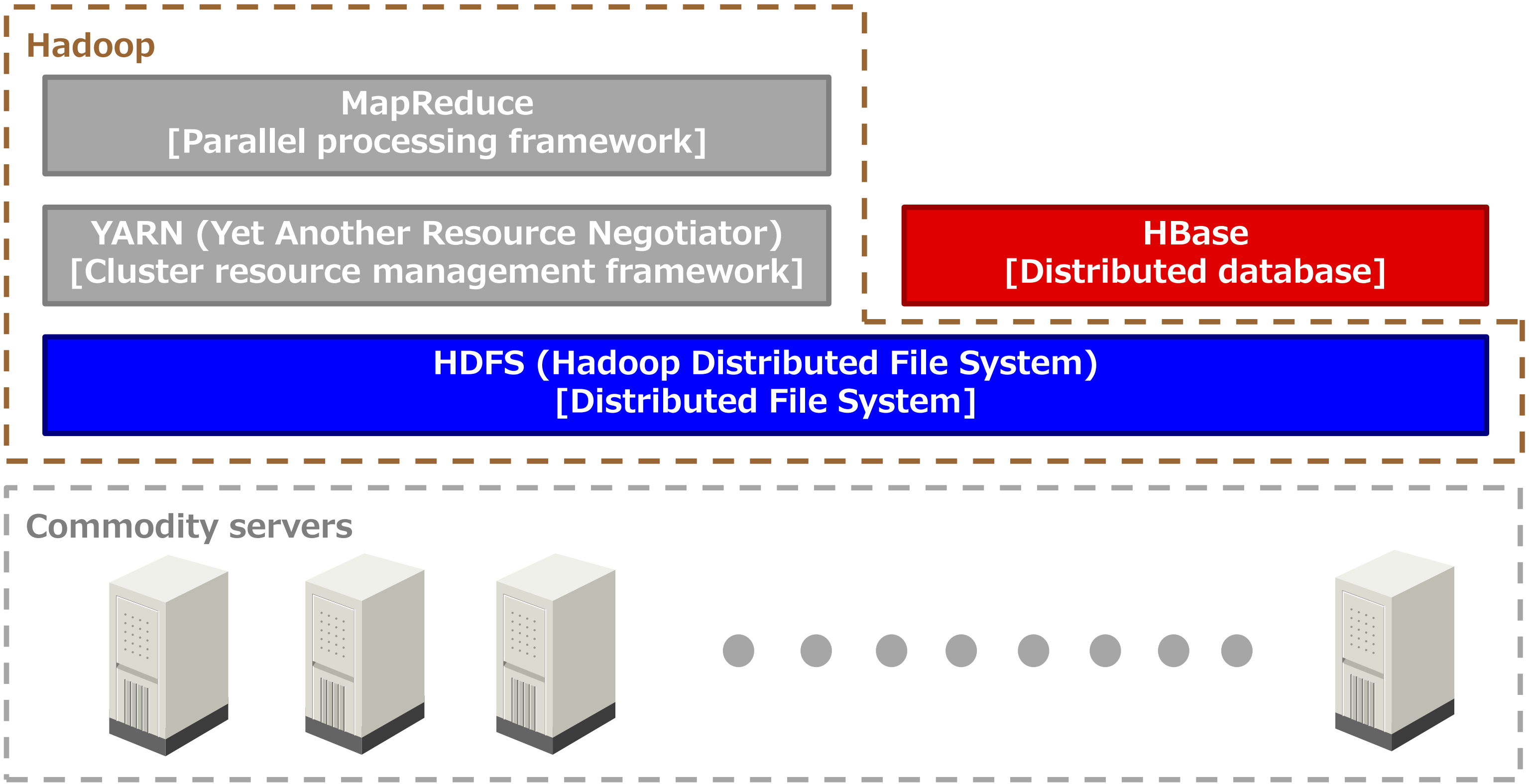
- A Google Bigtable clone.
  - Implemented in Java based on the paper of Bigtable.

- One of the OSS in Apache Hadoop eco-system.

Relationship between HBase and Hadoop（HDFS）

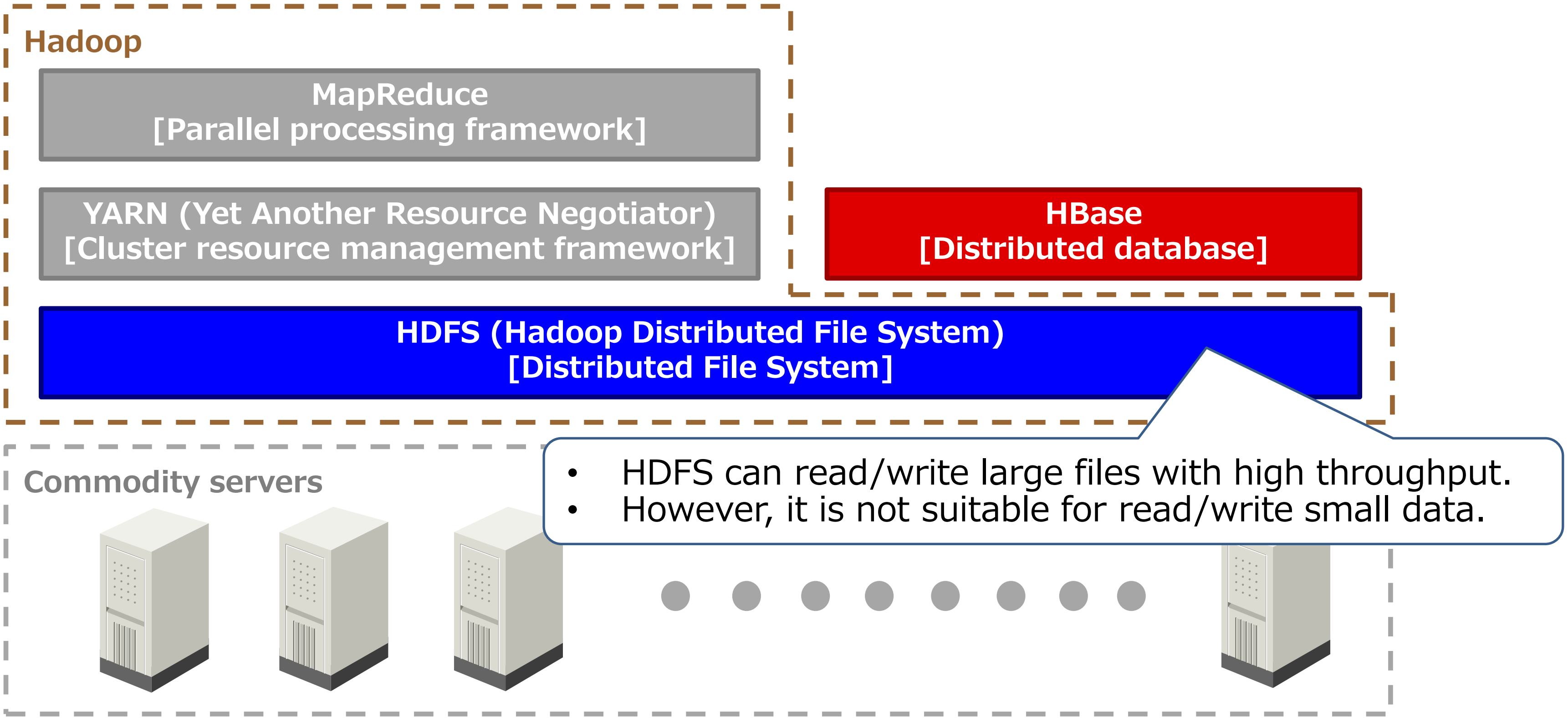
HITACHI
Inspire the Next

- HBase build on HDFS (Hadoop Distributed File System).



Relationship between HBase and Hadoop（HDFS）

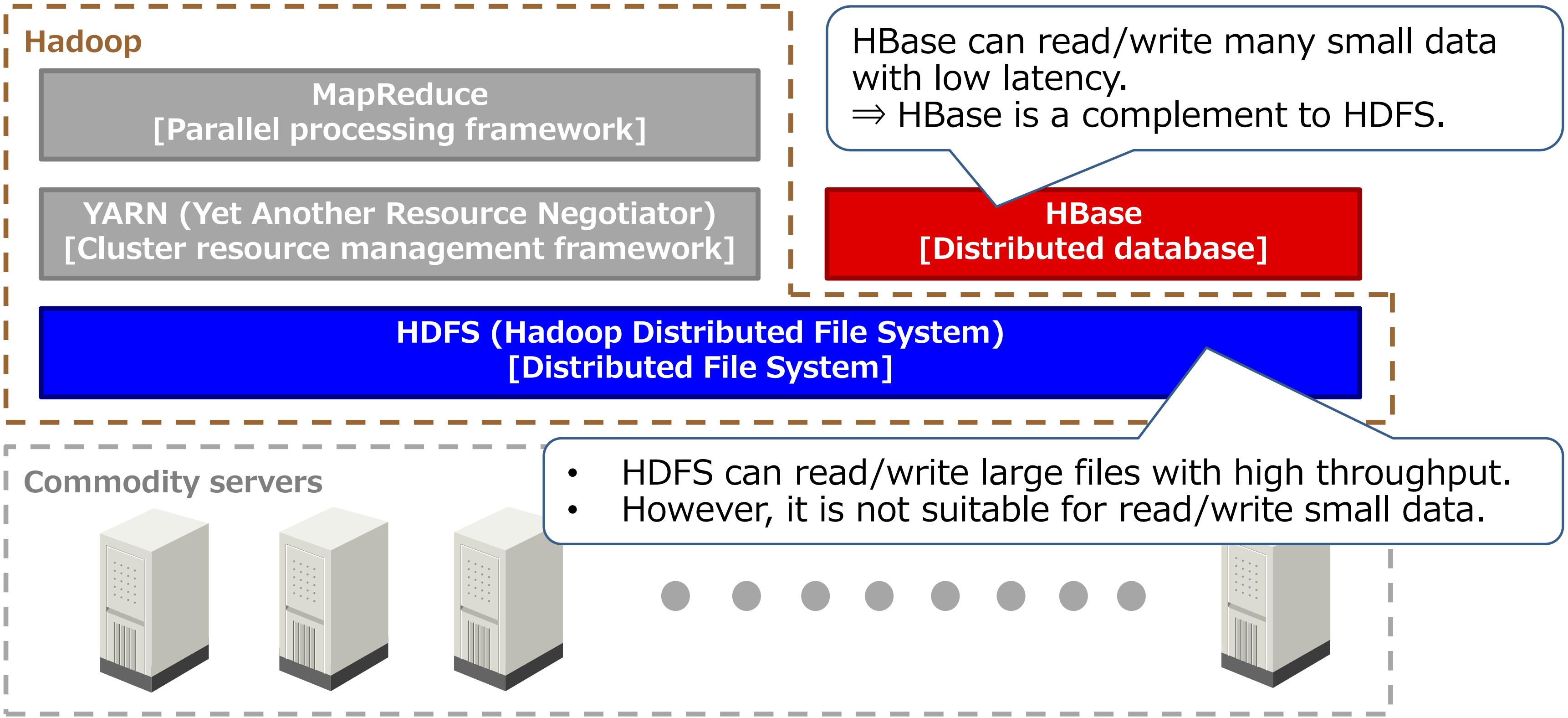
- HBase build on HDFS (Hadoop Distributed File System).



Relationship between HBase and Hadoop（HDFS）

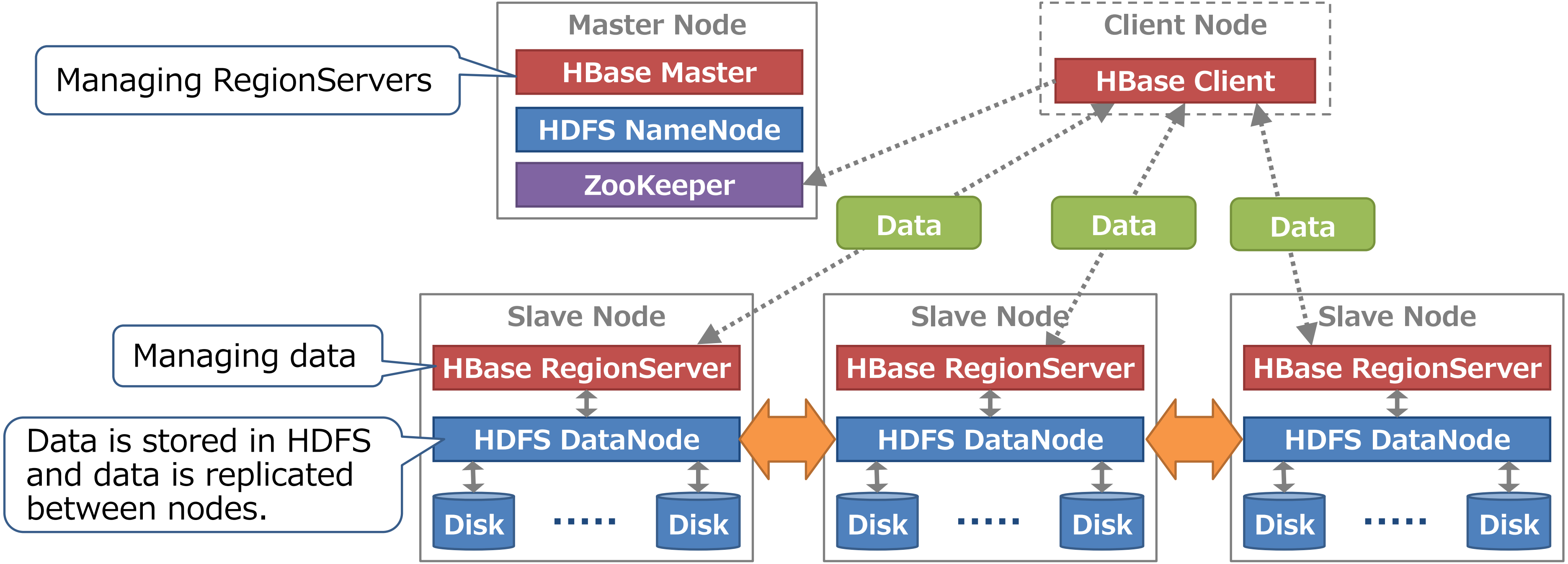
HITACHI
Inspire the Next

- HBase build on HDFS (Hadoop Distributed File System).

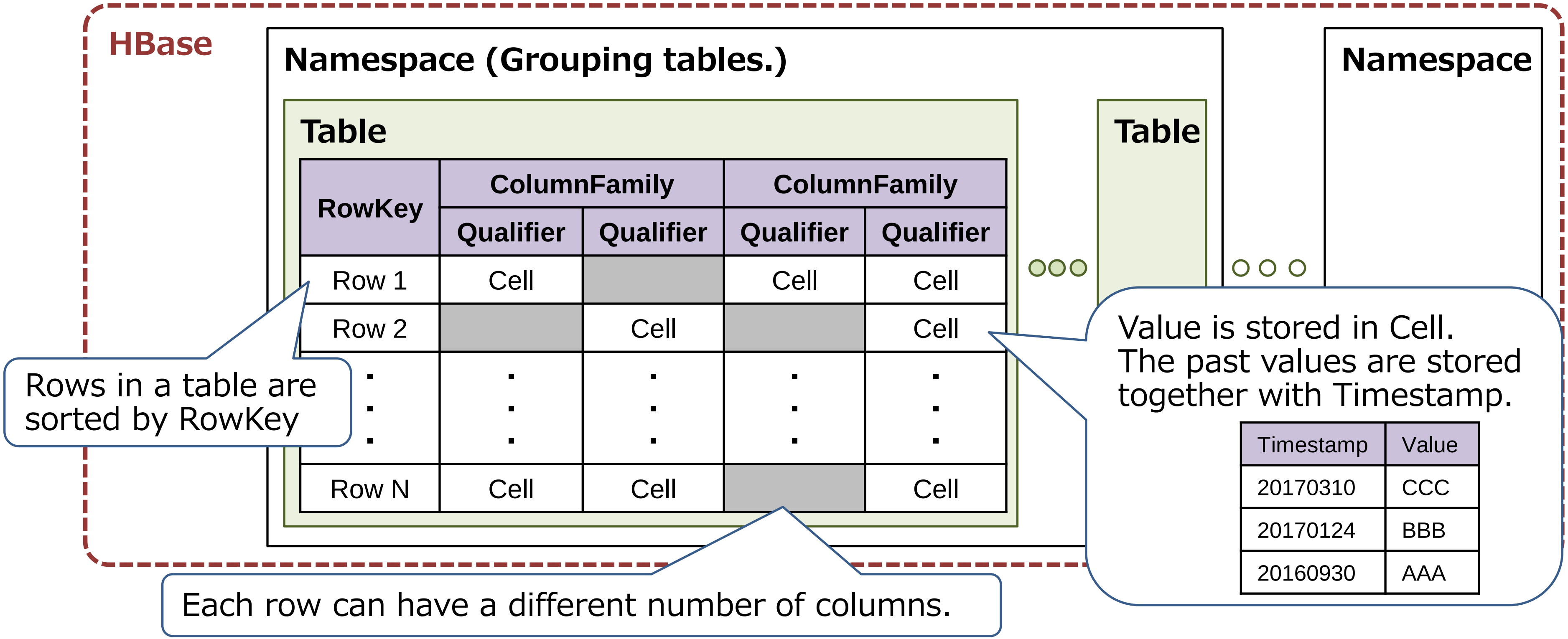


HBase architecture: Master/Slave model

- HBase processes the request and HDFS saves the data.



Data model: Conceptual view



RowKey	ColumnFamily		ColumnFamily	
	Qualifier	Qualifier	Qualifier	Qualifier
Row 1	Cell		Cell	Cell
Row 2		Cell		Cell
:	:	:	:	:
Row N	Cell	Cell		Cell

Timestamp	Value
20170310	CCC
20170124	BBB
20160930	AAA

- This table looks like a RDB’s table.

Data model: Physical view

HITACHI
Inspire the Next

- Data is stored as key value.
  - ➢ The keys are sorted in the order of **RowKey**, **Column** (ColumnFamily:qualifier), **Timestamp**.
  - ➢ It is a “multi-dimensional sorted map”.
    - SortedMap<**RowKey**, SortedMap<**Column**, SortedMap<**Timestamp**, **Value**>>>

Conceptual view of Table

RowKey	fam1		fam2	
	Col1	Col2	Col3	Col4
Row 1	-		Val_03	Val_04
Row 2	Val_05	Val_06	-	

Physical view of Table

Key Value

RowKey	Column (ColumnFamily:qualifier)	Timestamp	Type	Value
Row 1	fam1:Col1	20170310	Delete	-
Row 1	fam1:Col1	20170310	Put	Val_01
Row 1	fam2:Col3	20170215	Put	Val_03
Row 1	fam2:Col4	20170309	Put	Val_04
Row 2	fam1:Col1	20170310	Put	Val_05
Row 2	fam1:Col2	20160104	Put	Val_06
Row 2	fam2:Col3	20170221	Delete	-
Row 2	fam2:Col3	20170204	Put	Val_07

Operations and functions

- **Operations**
  - ➢ Put, Get, Scan, Delete, etc.
- **Functions**
  - ➢ Index
    - Only be set to RowKey and Column.
  - ➢ Transaction
    - Only within one Row.

Put a row

Get a row with random access

Scan multiple rows with sequential access

Delete a value by adding tombstones

RowKey	Column	Timestamp	Type	Value
Row 1	fam1:Col1	20170310	Delete	-
Row 1	fam1:Col1	20170310	Put	Val_01
Row 2	fam2:Col3	20170215	Put	Val_03
Row 2	fam2:Col4	20170309	Put	Val_04
Row 3	fam1:Col1	20170310	Put	Val_05
Row 3	fam1:Col2	20160104	Put	Val_06
Row 4	fam2:Col3	20170221	Delete	-
Row 4	fam2:Col3	20170204	Put	Val_07

Distributed data management

- How is a table physically divided?

Table

RowKey	Column	・・・	Value
Row 1	fam1:Col1	・・・	Val_01
Row 1	fam1:Col2	・・・	Val_02
Row 1	fam1:Col3	・・・	Val_03
Row 1	fam2:Col1	・・・	Val_04
Row 2	fam1:Col1	・・・	Val_05
Row 2	fam2:Col2	・・・	Val_06
Row 2	fam2:Col3	・・・	Val_07
Row 3	fam1:Col1	・・・	Val_08
Row 3	fam2:Col1	・・・	Val_09
Row 4	fam1:Col2	・・・	Val_10
Row 4	fam1:Col4	・・・	Val_11
Row 4	fam2:Col3	・・・	Val_12
Row 4	fam2:Col5	・・・	Val_13

Table is divided into Region with the range of RowKey

Table

Region (Row1-2)

RowKey	Column	･･･	Value
Row 1	fam1:Col1	･･･	Val_01
Row 1	fam1:Col2	･･･	Val_02
Row 1	fam1:Col3	･･･	Val_03
Row 1	fam2:Col1	･･･	Val_04
Row 2	fam1:Col1	･･･	Val_05
Row 2	fam2:Col2	･･･	Val_06
Row 2	fam2:Col3	･･･	Val_07

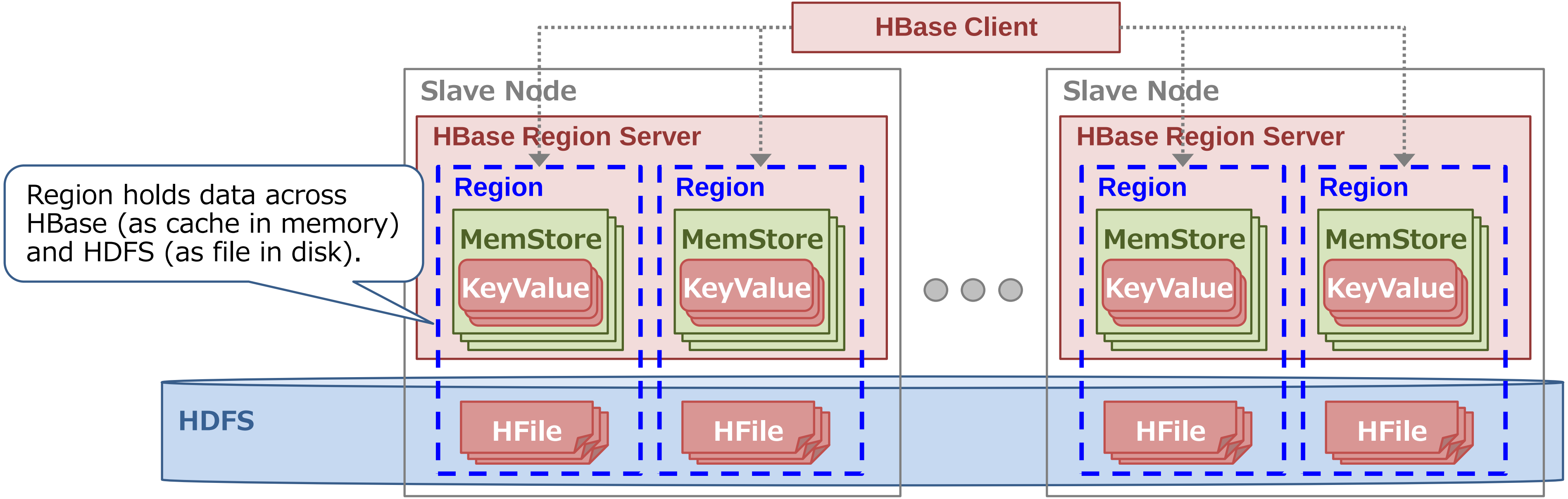
Region (Row3-4)

Row 3	fam1:Col1	･･･	Val_08
Row 3	fam2:Col1	･･･	Val_09
Row 4	fam1:Col2	･･･	Val_10
Row 4	fam1:Col4	･･･	Val_11
Row 4	fam2:Col3	･･･	Val_12
Row 4	fam2:Col5	･･･	Val_13

Data is distributed on the cluster via Regions

HITACHI
Inspire the Next

- Automatic sharding
  - Regions are automatically split and re-distributed as data grows.
- Simple horizontal scaling
  - Adding slave nodes improves performance and expands disk capacity.



Summary of HBase architecture

- Simple horizontal scaling
  - ➢ Adding slave nodes improves performance and expands disk capacity
- Data is stored as sorted key value
  - ➢ Like multi-dimensional sorted map.
  - ➢ By designing RowKey carefully, data that are accessed together are physically co-located.
- Limited the index and transaction
  - ➢ Index : Only be set to RowKey and Column.
  - ➢ Transaction: Only within one Row.

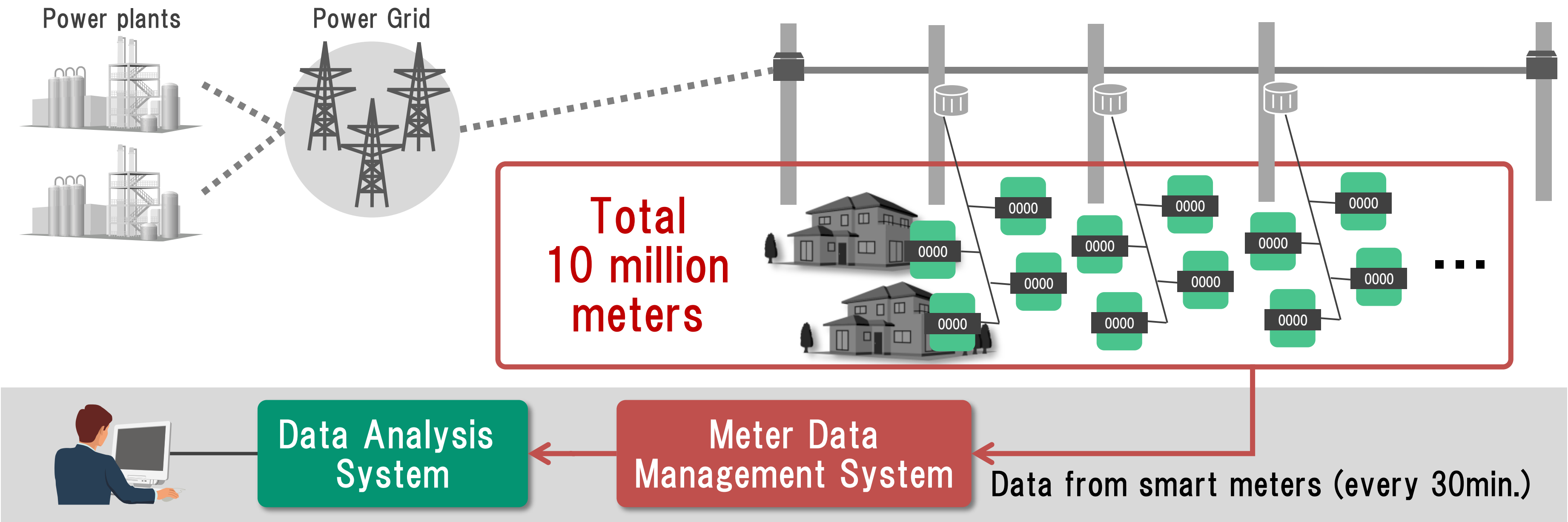
4. Performance evaluation with 10 million smart meter data

i. Evaluation scenario

Smart meter data management

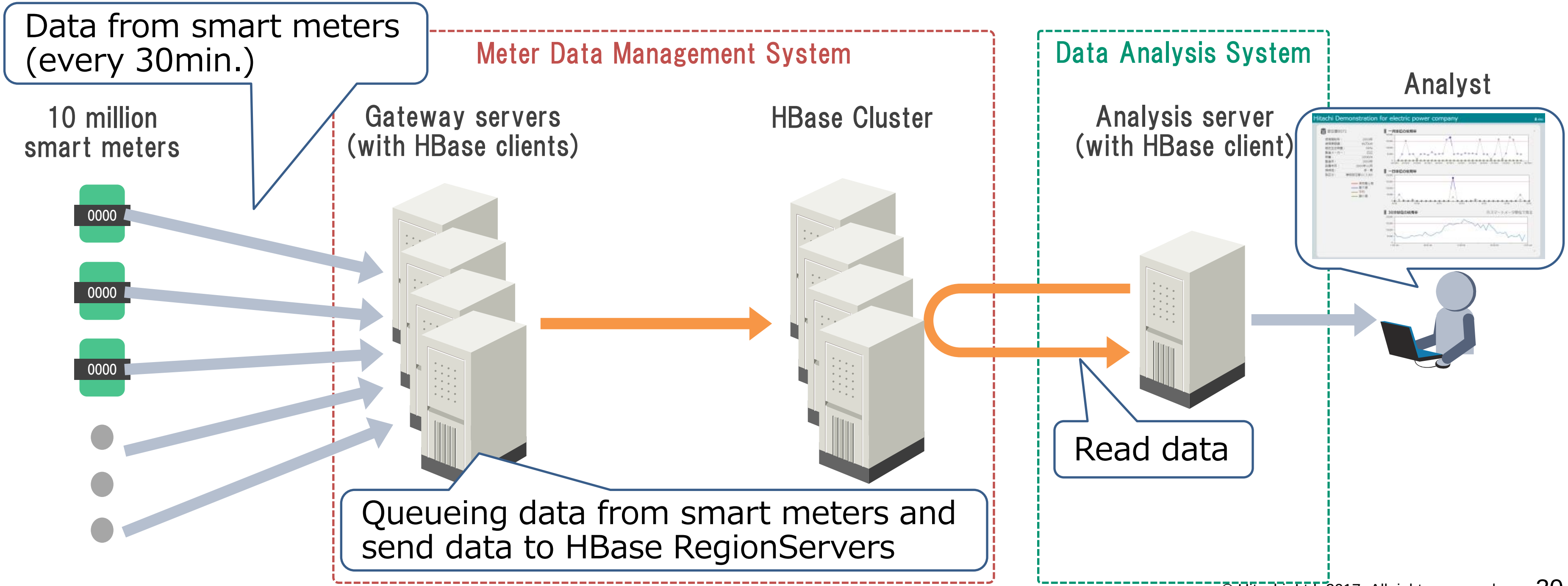
HITACHI Inspire the Next

- We assumed the Meter Data Management System for 10 million smart meters.
  - ➢ Smart meters collect consumption of electric energy from customers.
    - Send the collected data to the Meter Data Management System every 30 minutes.
  - ➢ The collected data is used for power charge calculation and demand forecast analysis, etc.



System overview

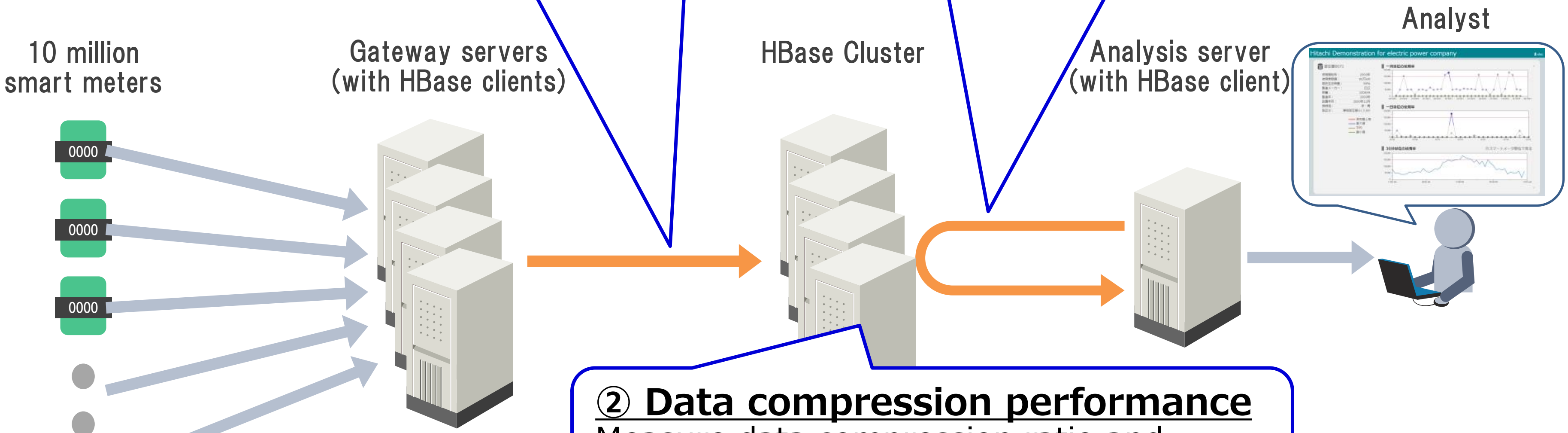
- Write 10 million records every 30 minutes in HBase.
- Read to analyze records stored in HBase.



Contents of performance evaluation

① Write performance
Measure write time and throughput of 10 million records.

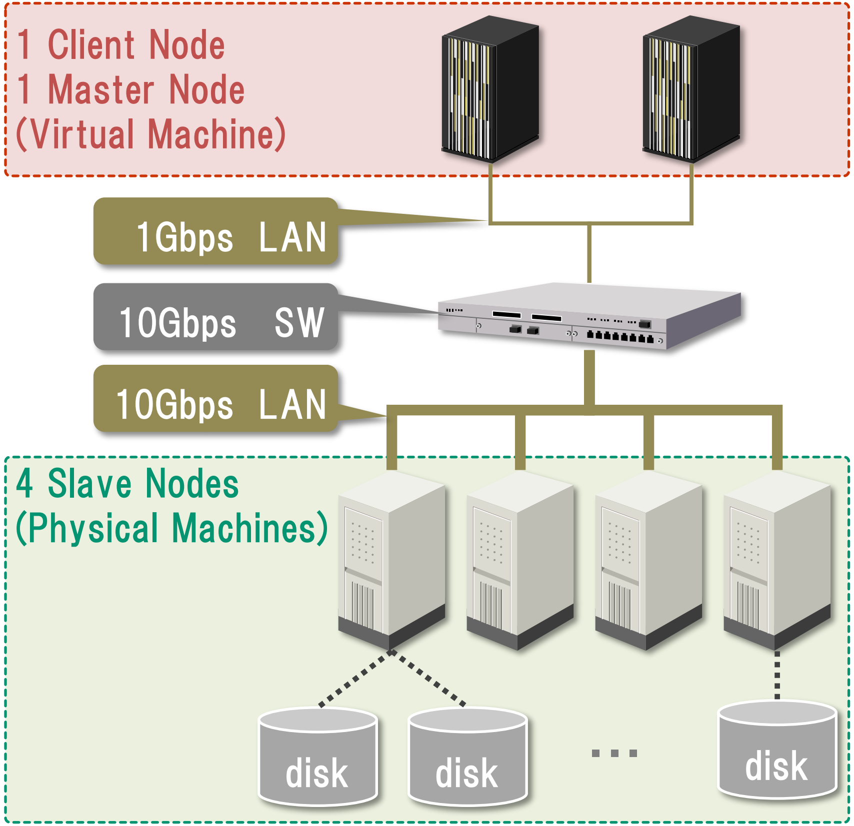
③ Read performance
Measure read time and throughput in two kinds of analysis use cases.



② Data compression performance
Measure data compression ratio and compression / decompression time.

Evaluation environment

Software version
CDH5.9 (HBase1.2.0 + Hadoop2.6.0)



	Client Node	Master Node
CPU Core	16	2
Memory	12 GB	16 GB
# of disk	1	1
Capacity of disk	80 GB	160 GB

	Per slave node	Total
CPU Core	32	128
Memory	128 GB	512 GB
# of disk	6	24
Capacity of disk	900 GB	-
Total capacity of disks	5.4 TB (5,400 GB)	21.6 TB (21,600 GB)

Table design

HITACHI
Inspire the Next

- Divided the table into 400 Regions in advance.
  - 100 Regions per RegionServer
  - Region split key: 0001, 0002, …, 0399

Region (～0001) | Region (0001～0002) | Region (0002～0003) | ○○○ | Region (0399～)

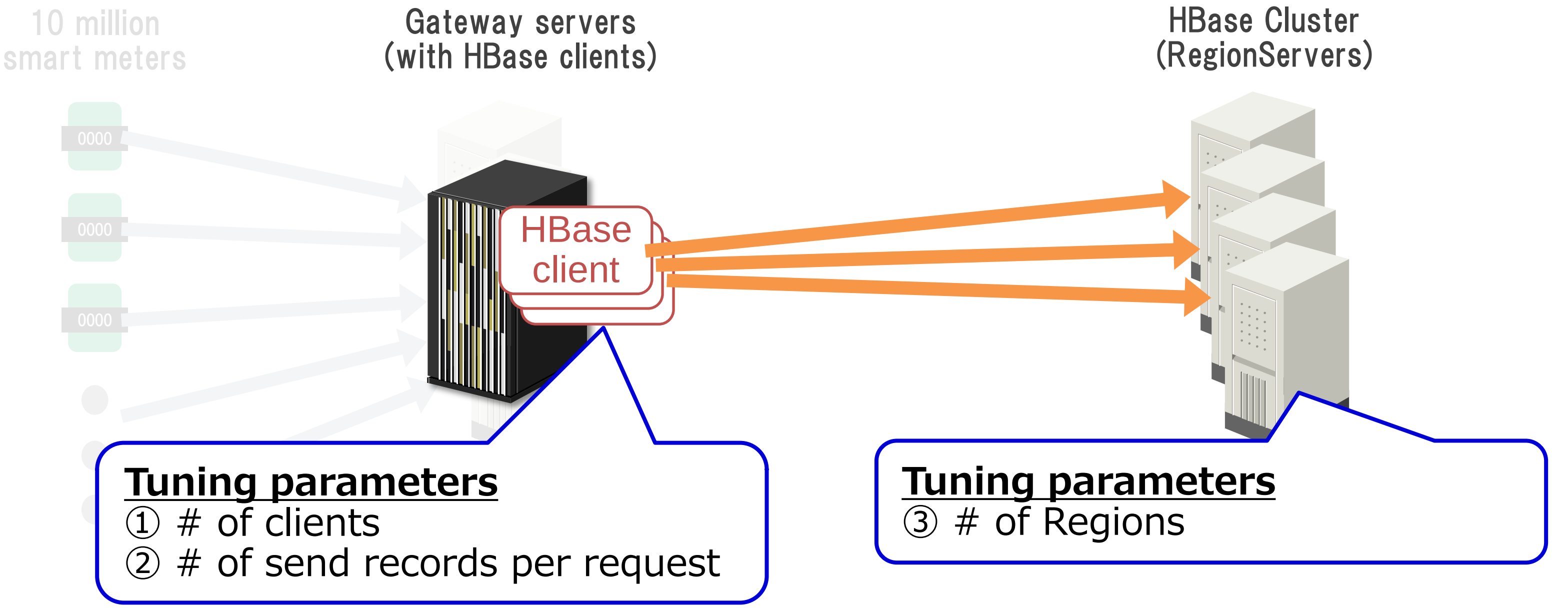
To distribute data among Regions, add 0000 to 0399 (meter ID modulo 400) to the head of RowKey. This technique is called "Salt".

RowKey (<Salt>-<Meter ID>-<Date>-<Time>)	Column (ColumnFamily:qualifier)	Timestamp	Type	Value
0000-0000000001-20170310-1100	CF:		Put	3.241
0000-0000000001-20170310-1030	CF:		Put	0.863
⋯	⋯		Put	0.430
0000-0000000001-20160910-1100	CF:		Put	0.044
0001-0000000002-20170310-1100	CF:		Put	2.390
⋯	⋯		Put	1.432

ii. Evaluation of write performance

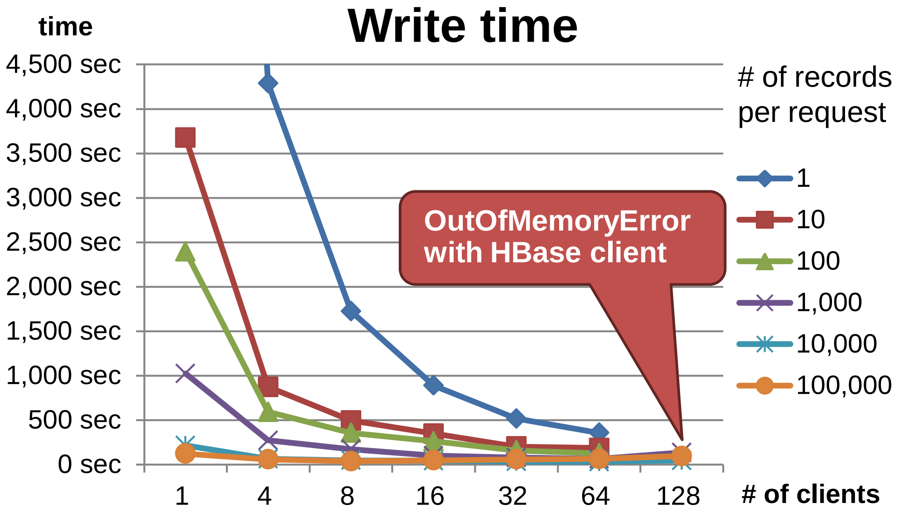
Evaluation of write performance

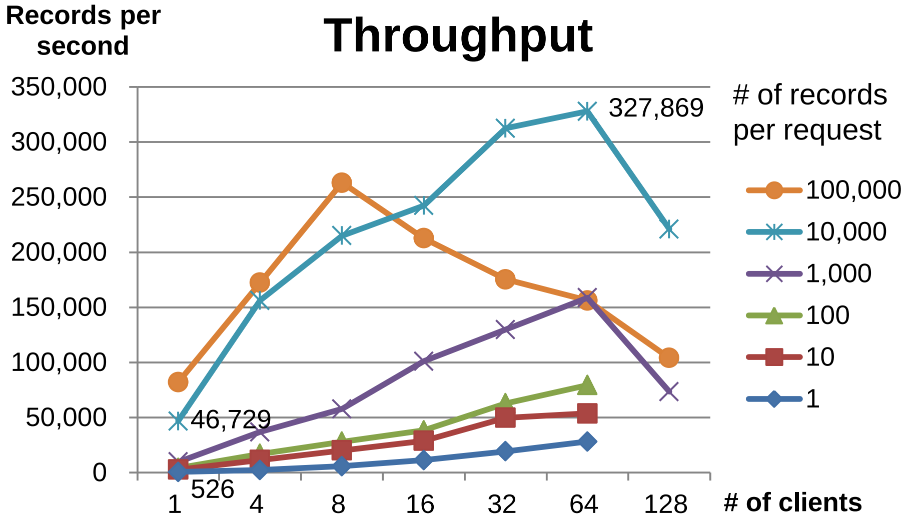
- Generate 10 million records with HBase clients.
- Send put request using multi clients.
- Measured the write time and throughput of 10 million records.



Write performance

- Write time and throughput of 10 million records.





- Stored multiple records by one request:
  - ➢ Records per request: 1 to 10,000 ⇒ Throughput: 526 to 46,729 records/sec (89x)
- Increased the number of clients:
  - ➢ # of Clients: 1 to 64 ⇒ Throughput: 46,729 to 327,869 records/sec (7x)

iii. Evaluation of Compression performance

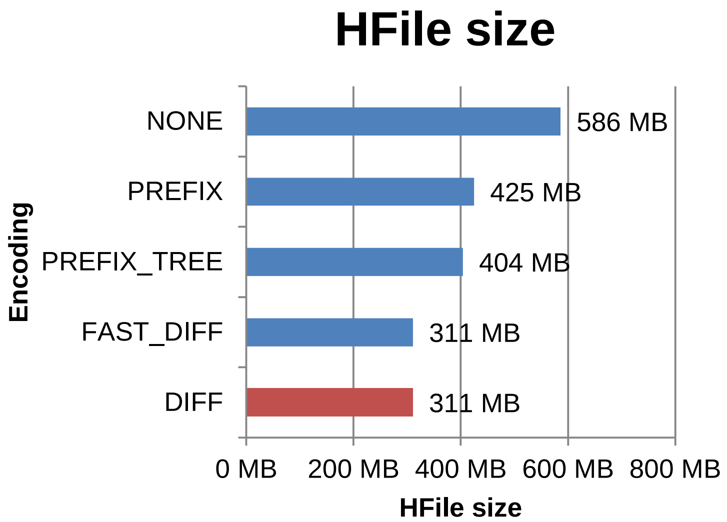
Compressor and data block encoding

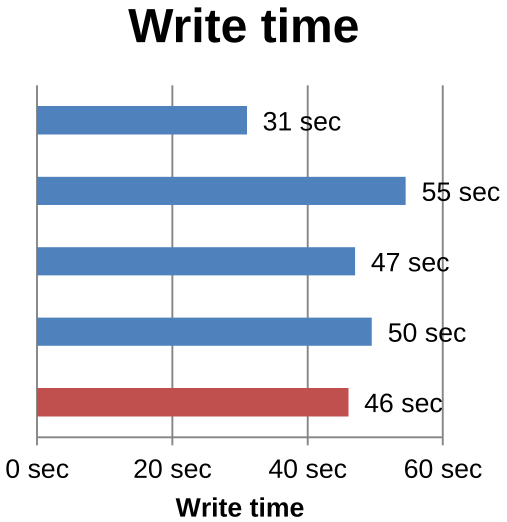
- HBase tends to increase data size for the following reasons.
  - The number of records increases because data is stored in key value format.
  - Each record length is long because a key is composed of many fields.

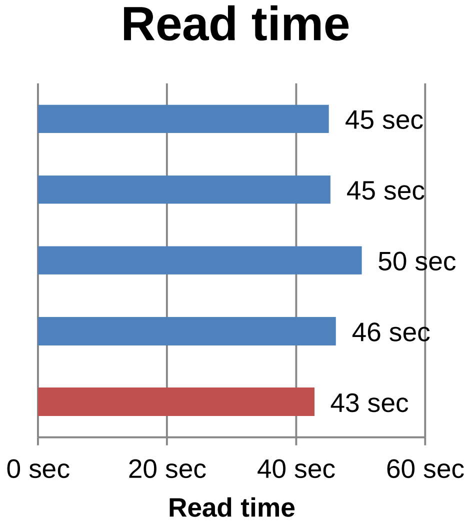
- Compress data with a combination of compressor and data block encoding.

- Measured the file size, write time, and read time of 10 million records.

Data block encoding performance with 10 million records







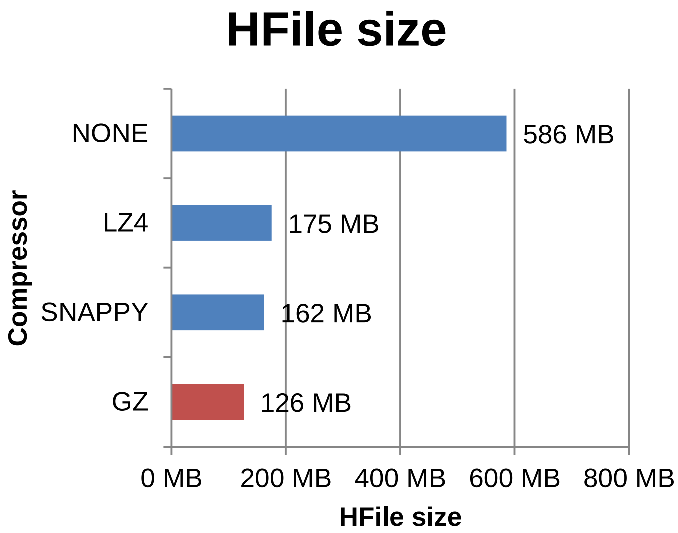
Reduced to 53%
by DIFF encoding

Increased 48%
by DIFF encoding

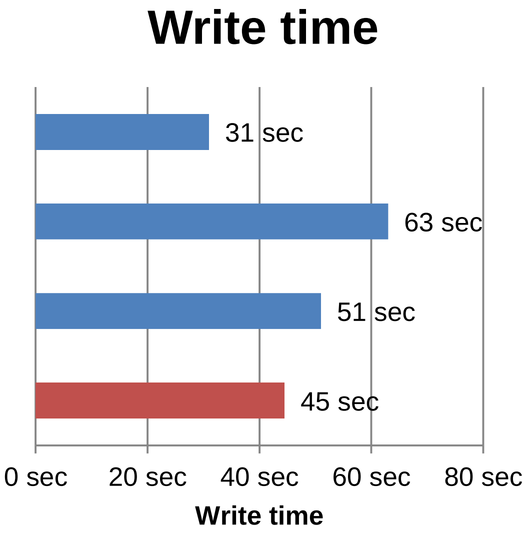
Reduced 4%
by DIFF encoding

Compressor performance with 10 million records

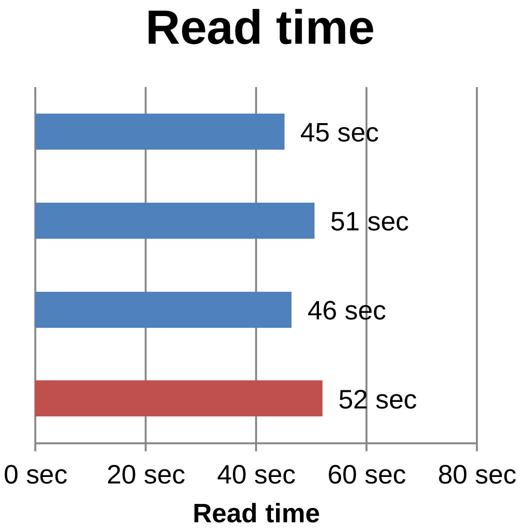
HITACHI
Inspire the Next



Reduced to 22%
by GZip algorithm

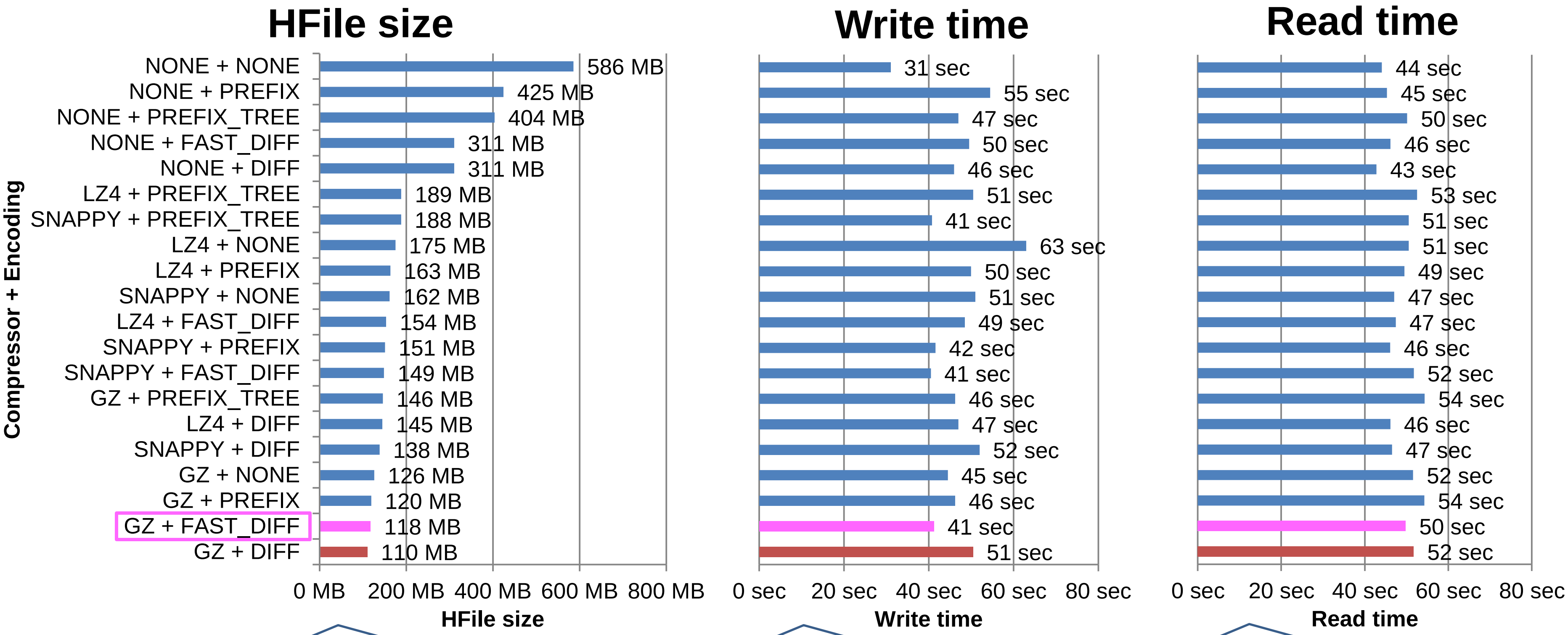


Increased 68%
by GZip algorithm



Increased 15%
by GZip algorithm

Compressor and data block encoding performance with 10 million records



Reduced to 19%
by GZip + FAST_DIFF

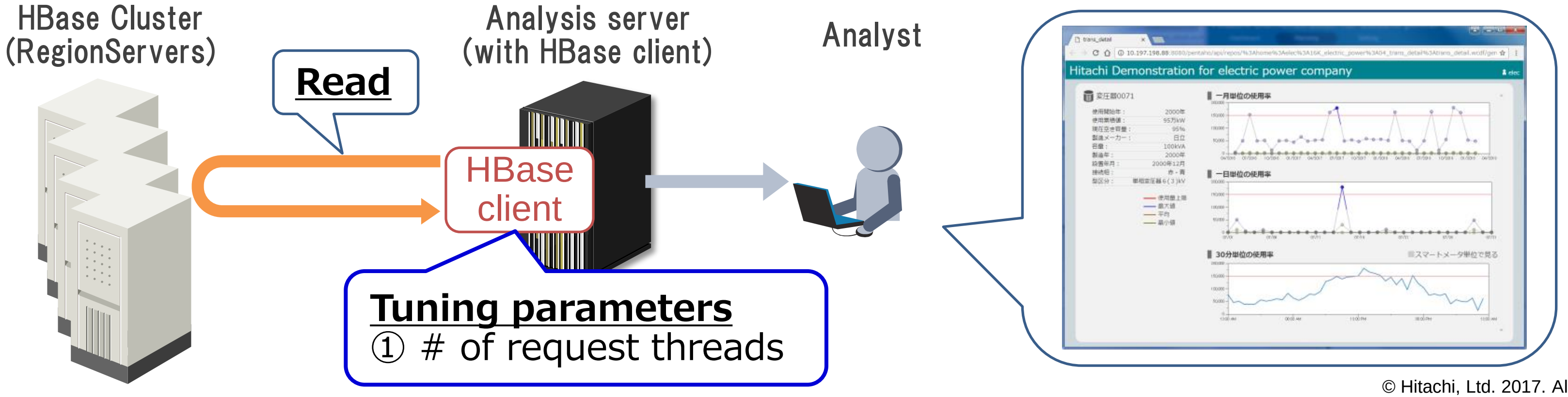
Increased 33%
by GZip + FAST_DIFF

Increased 14%
by GZip + FAST_DIFF

iv. Evaluation of read performance

Evaluation of read performance

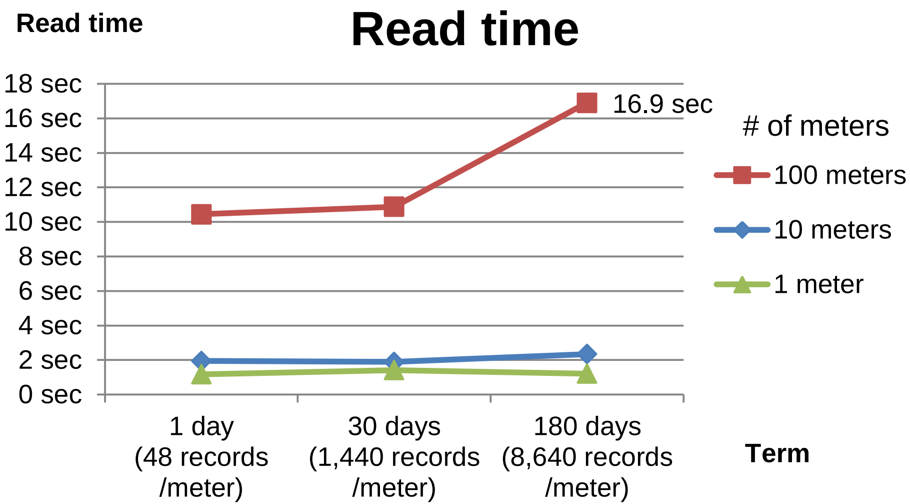
- Measure the read time and throughput in two kinds of analysis use cases.
  - ➢ **<u>Use case A</u>**: Scan time series data of a few meters.
    - To display the transition of power consumption per meter in the line chart.
  - ➢ **<u>Use case B</u>**: Get the latest data of many meters.
    - To calculate the average and total value of the latest power consumption.
  - ➢ Evaluation settings
    - Dataset: 10 million meter * 180 days records (Compressed by FAST_DIFF + GZ)
    - Disabled caches and make sure to read data from disk.

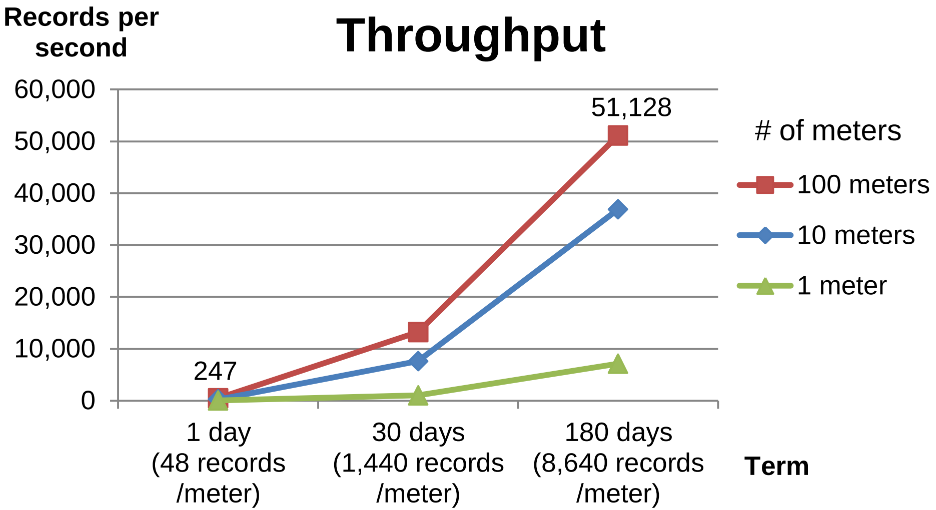


Use case A: Scan time series data of a few meters

HITACHI
Inspire the Next

- Scan meter data for **1-180 days** of **1-100 meters**.
  - ➢ Scan time series data of one meter by one scan.



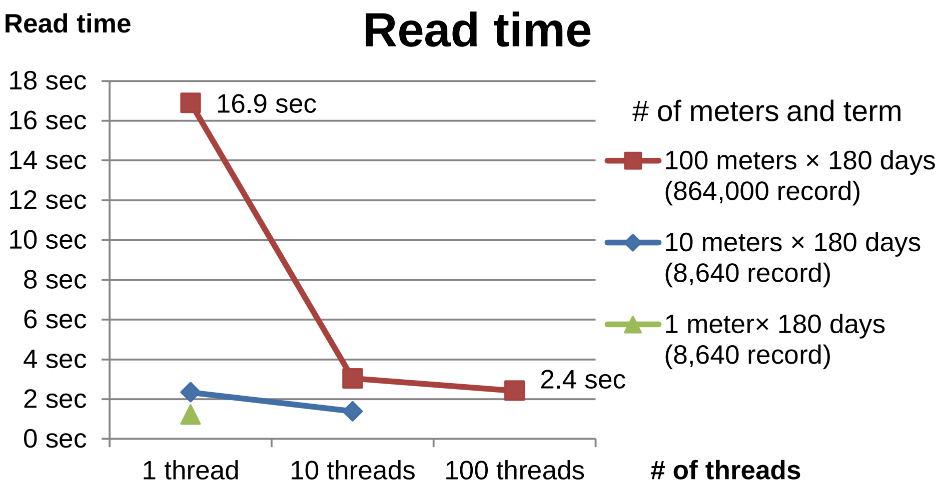


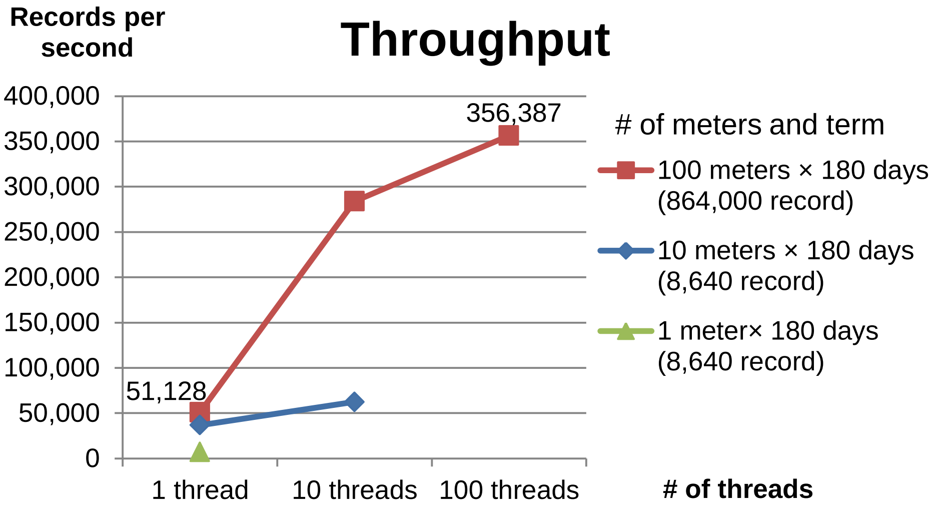
Since read multiple data with one Scan, the throughput improves as the term was longer.

➢ Term: 1 to 180 days ⇒ Throughput: 247 to 51,128 records/sec (207x)

Use case A: Scan time series data of a few meters (with multi thread)

- Scan meter data for **180 days** of **1-100 meters**.
  - ➢ Scan request was executed in multi thread. (Maximum 1 Scan 1 thread)



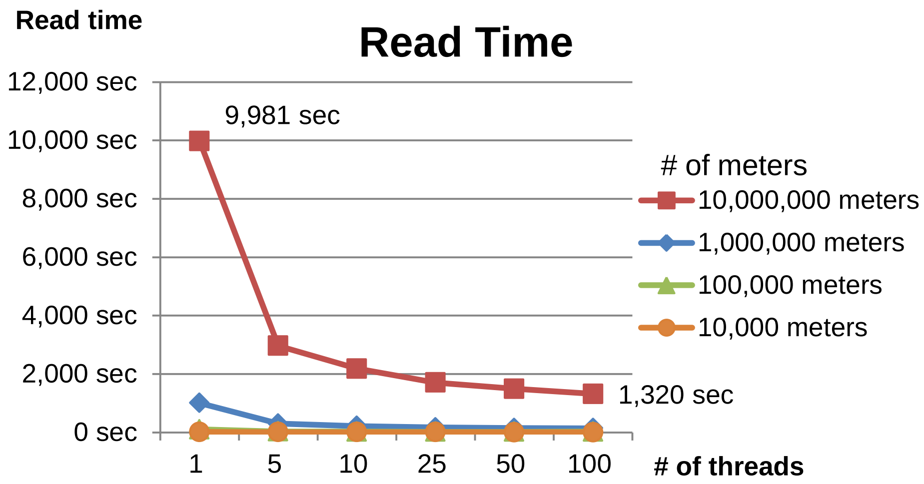


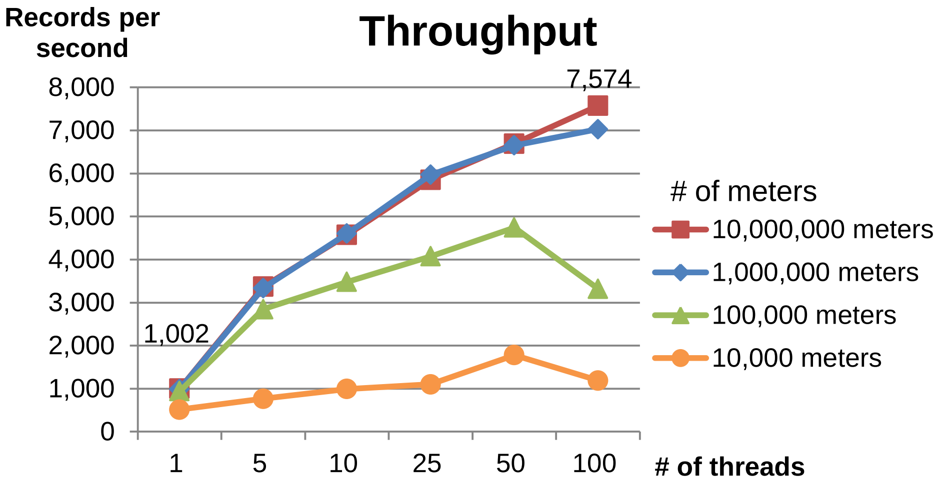
Throughput was improved by running Scan requests in parallel.
- ➢ # of threads: 1 to 100 ⇒ Throughput: 51,128 to 356,387 records/sec (7x)

Use case B: Get the latest data of many meters (with multi thread)

HITACHI Inspire the Next

- Get the latest time (30 minutes) data of **10,000 to 10 million meters**.
  - ➢ Scan request can not be applied to these data.
  - ➢ Requests are executed in multi thread.
  - ➢ Batch execution of multiple “Get” request by one “batch” request.





Throughput was improved by running Get requests in parallel.

➢ # of threads: 1 to 100 ⇒ Throughput: 1,002 to 7,574 records/sec (7.5x)

Comparison of Scan request with Get request

HITACHI
Inspire the Next

Use case A:
Scan 180 days time series data of 100 meters with 100 thread.
= **Throughput 356,387 records/second**

Use case B:
Get the latest 30 min. data of 10,000,000 meters with 100 thread.
= **Throughput 7,574 records/second**

RowKey (<Salt>-<Meter ID>-<Date>-<Time>)	…	Value
0000-0000000001-20170310-1100		3.241
0000-0000000001-20170310-1030		0.863
…		…
0000-0000000001-20160910-1100		0.044
…		…
0200-0000000201-20170310-1100		10.390
0200-0000000201-20170310-1030		14.325
…		…
0200-0000000201-20160910-1100		9.32
…		…

- Scan request's throughput was about 47x higher than the Get request.
- Careful RowKey design is important.
  - Place the data that are accessed together physically co-located.

5. Summary

Summary

- HBase is suitable for storing time series data generated by sensor devices.

- Lessons from performance evaluation:
  - ➢ Careful RowKey design to be able to scan data is important.
    - Scan request‘s throughput was more than 47x that of Get request.
  - ➢ HBase has high multi-client / multi-thread concurrency.
    - Throughput of the Put / Scan / Get request with multi-client / multi-thread is 7x faster than single-client / single-thread.
  - ➢ Choosing the appropriate compression setting.
    - The storage size of time series data could be reduced to 19%.

Trademarks

- Apache HBase and Apache Hadoop are either a registered trademark or a trademark of Apache Software Foundation in the United States and/or other countries.
- Other company and product names mentioned in this document may be the trademarks of their respective owners.